1. 样例一

```
输入是: 764312805
输出是: 420583716
```

```
BFS的搜索顺序是:
764312805
764302815
704362815
740362815
742360815
742306815
742036815
742836015
742836105
742836150
742830156
742803156
742083156
042783156
402783156
482703156
482753106
482753016
482053716
482503716
402583716
420583716
BFS cost 688ms
```

```
DFS的搜索顺序是:
764312805
764302815
704362815
740362815
742360815
742306815
742036815
742836015
742836105
742836150
742830156
742803156
742083156
042783156
402783156
482703156
482753106
482753016
482053716
482503716
402583716
420583716
DFS cost 8426ms
```

```
HS的搜索顺序是:
764312805
764302815
704362815
740362815
742360815
742306815
742036815
742836015
742836105
742836150
742830156
742803156
742083156
042783156
402783156
482703156
482753106
482753016
482053716
482503716
402583716
420583716
HS cost 63ms
```

2. 样例二

```
输入是： 704182653
输出是： 287145306
```

```
BFS的搜索顺序是：
704182653
740182653
742180653
742183650
742183605
742103685
742130685
740132685
704132685
074132685
174032685
174302685
174320685
170324685
107324685
127304685
127384605
127384065
127084365
027184365
207184365
287104365
287140365
287145360
287145306
BFS cost 1138ms
```

```
DFS的搜索顺序是：
704182653
740182653
742180653
742183650
742183605
742103685
742130685
740132685
704132685
074132685
174032685
174302685
174320685
170324685
107324685
127304685
127384605
127384065
127084365
027184365
207184365
287104365
287140365
287145360
287145306
DFS cost 32365ms
```

```
HS的搜索顺序是：
704182653
074182653
174082653
174802653
174820653
170824653
107824653
127804653
127084653
027184653
207184653
287104653
287140653
287143650
287143605
287143065
287043165
287403165
287430165
287435160
287435106
287405136
287045136
287145036
287145306
HS cost 225ms
```

3. 样例三

```
输入是: 504283716
输出是: 725681043
```

```
BFS的搜索顺序是:
504283716
584203716
584213706
584213760
584210763
584201763
504281763
540281763
541280763
541208763
501248763
051248763
251048763
251748063
251748603
251708643
251780643
250781643
205781643
025781643
725081643
725681043
BFS cost 732ms
```

```
DFS的搜索顺序是:
504283716
584203716
584213706
584213760
584210763
584201763
504281763
540281763
541280763
541208763
501248763
051248763
251048763
251748063
251748603
251708643
251780643
250781643
205781643
025781643
725081643
725681043
DFS cost 8935ms
```

```
HS的搜索顺序是:
504283716
584203716
584213706
584213760
584210763
584201763
504281763
540281763
541280763
541208763
501248763
051248763
251048763
251748063
251748603
251708643
251780643
250781643
205781643
025781643
725081643
725681043
HS cost 61ms
```

4. 算法分析与性能评估

该代码同时实现了 BFS,DFS,HS 三种搜索方式。

- BFS 广度搜索算法采用 while 循环，以队列 q 非空作为循环跳进，每次循环从队列 q 中取出一个状态作为当前状态，把当前状态的所有可能的下一个状态全部添加到队列 q 中，在添加到队列 q 之前，还需要利用 map 容器 mp 检查该状态是否已经搜索过了，如果已经搜索过了，则不添加，如果没有搜索过，则进行添加操作。如此反复，如果发现下一个状态是目标状态，则查询成功，循环结束，打印即可；如果队列 q 为空的时候还没有查询成功，则认为查询失败，退出循环。
- DFS 深度搜索算法递归调用 dfs 函数，初始设置最大深度 max_depth 为 1，每一次 while 循环 max_ 就加 1，进入 dfs 函数，获取当前状态的下一个状态，然后以下一个状态为参数递归调用 dfs 函数，直到找到目标状态返回 true 或者当前深度超过最大深度返回 false
- Heuristic search 算法采用与 BFS 广度搜索算法类似的思路，只不过存储状态的队列是优先队列 pq，该队列以二元组 pair<int,int> 为单元进行存储，每一个存储单元 pair<int,int> 的前一个 int 表示该状态的估价值（即该状态的深度和该状态与目标状态差值的和），后一个 int 表示该状态在 str_arr 数组和 step 数组中的下标定义 pq 时采用小根堆的排序方式，这样就可以保证每一次从 pq 中取出的状态都是 pq 中估价值最小的状态
- 通过三个算法运行样例一、二、三的输出结果我们可以发现，Heuristic search 算法的效率明显是最高的，其次是 BFS 广度搜索算法，最慢的是 DFS 深度搜索算法
- 广度优先搜索（BFS）：优点：确保找到最短路径（如果存在）。当目标状态位于较浅的层级时，通常效果较好。缺点：需要存储每个已访问节点的状态，因此对内存要求较高。在搜索树较深或分支因子较大时，可能会占用大量内存和时间。不适用于无限状态空间的情况。
- 深度优先搜索（DFS）：优点：占用内存较少，因为它只需要存储当前路径上的节点。在搜索树较深或分支因子较大时，可能比较快速。缺点：不保证找到最短路径，可能会陷入无限循环或者搜索到无意义的路径。当目标状态位于较深的层级时，效率可能较低。受限于递归深度，可能会导致堆栈溢出。DFS 可能陷入局部最优，即搜索进度似乎毫无进展的区域。
- 启发式搜索（Heuristic Search）：优点：通过引入启发式函数，可以在搜索过程中更加智能地选择下一步的节点，提高搜索效率。可以在较短的时间内找到解决方案。缺点：受启发式函数质量的影响，如果启发式函数不好，可能会导致搜索效率低下。有可能陷入局部最优解，无法找到全局最优解。对于复杂的问题，设计合适的启发式函数可能会很困难。

5. 源代码

```
#define _CRE_SECURE_NO_WARNINGS
#define PAIR pair<int,int>
#include<iostream>
#include<map>
#include<unordered_map>
#include<queue>
#include<algorithm>
#include<cmath>
#include<cstring>
#include<ctime>
using namespace std;
const int N = 362880 + 7;
//定义clock_t变量，用于记录程序运行时间
clock_t  Begin, End;

/*map容器mp,用来查重，将已经搜索过的状态放入mp中，之后得到新的状态在mp
中查询看是不是已经被搜索过。
前一个string表示当前状态，后一个string表示当前状态的前一个状态(即当前
状态是由哪一个状态转变而来的)*/
unordered_map<string, string> mp;

/*优先队列pq，在启发式搜索中用来存储每个状态的参数，pair前一个int表示
估价函数，后一个int表示其在step数组和str_arr数组中的下标
利用优先队列堆排序的特性，每次将最小的pair<int,int>排在队列队首，
我们每次从pq中取出来的对应的string,其估价函数是最小的*/
//将PAIR宏定义为pair<int,int>
priority_queue<PAIR, vector<PAIR>, greater<PAIR>> pq;

//队列q用于BFS搜索中存储状态
queue<string>q;

/*全局变量temp，用于DFS中调用s_push时记录更改后的状态，sour和dest
分别表示源状态和目标状态*/
string  temp, sour = "012345678", dest = "012345678", str_arr[N];

/*state表示是否查询成功，在BFS和启发式搜索中判断q_push和pq_push
```

```
是否找到目标状态。cnt用来表示str_arr数组的大小*/
int state = 0, cnt = 0, step[N], max_depth = 1;

/*step数组用来在启发式搜索中记录下标对应的状态相比与源状态走了多少步，
max_depth表示DFS的最大搜索深度
diff函数用来计算启发式搜索中源状态和目标状态的差异值*/
int diff(string str)
{
    int len = str.length(), res = 0;
    for (int i = 0; i < len; i++)
        if (str[i] != dest[i])
            res++;
    return res;
}

/*str表示要更改的字符串，pos表示str中'0'所在的下标
n表示'0'要跟自己哪边的值进行交换，例如当n为1时，表示str[pos]要跟
str[pos+1]交换位置(即'0'与他右边的值进行交换）*/
void q_push(string str, int pos, int n)
{
    string temp = str;
    char t = temp[pos + n];
    temp[pos + n] = '0';
    temp[pos] = t;
    //当temp == dest成立时表示查询成功，令state为1
    if (temp == dest)
        state = 1;
    /*用于查重，如果if语句成立则表示temp状态还没有搜索过，则将temp
    加入到mp和q中,并将str设置为temp的前一个状态*/
    if (mp.find(temp) == mp.end())
    {
        mp[temp] = str;
        q.push(temp);
    }
}

//str表示要更改的字符串，pos表示str中'0'所在的下标
/n表示'0'要跟自己哪边的值进行交换，例如当n为1时，表示str[pos]
要跟str[pos+1]交换位置(即'0'与他右边的值进行交换）*/
```

```
//temp表示str更改后的值，采用参数引用，返回源函数后temp就是更改后的状态
bool s_push(string str, int pos, int n, string& temp)
{
    temp = str;
    char t = temp[pos + n];
    temp[pos + n] = '0';
    temp[pos] = t;
    //用于查重，如果if语句成立则表示temp状态还没有搜索过，返回true即可
    if (mp.find(temp) == mp.end())
        return true;
    return false;
}

//str表示要更改的字符串，pos表示str中'0'所在的下标
/*n表示'0'要跟自己哪边的值进行交换，例如当n为1时，表示str[pos]
要跟str[pos+1]交换位置(即'0'与他右边的值进行交换) */
//s用来记录当前状态相比与源状态走了多少步，并将其存到step数组对应的位置
void pq_push(string str, int pos, int n, int s)
{
    string temp = str;
    char t = temp[pos + n];
    temp[pos + n] = '0';
    temp[pos] = t;
    if (temp == dest)
        state = 1;
    /*用于查重，如果if语句成立则表示temp状态还没有搜索过，则将temp
    加入到mp，pq和str_arr中，并在step数组中记录当前的步数*/
    if (mp.find(temp) == mp.end())
    {
        mp[temp] = str;
        step[cnt] = s;
        str_arr[cnt] = temp;
     /s表示步数，diff函数表示当前状态和目标状态的差异，二者加起来
     就是估价函数f(x),cnt表示当前状态在str_arr和step中的下标*/
        pq.push(pair<int, int>(s + diff(temp), cnt));
        cnt++;
    }
}
```

```
//str表示当前状态，depth表示当前搜索深度
bool dfs(string str, int depth)
{
//如果当前搜索深度大于最大搜索深度，则直接返回false
    if (depth > max_depth)
        return false;
//if语句成立表示查询成功，利用mp函数的定义，打印出搜索路径上的每一个状态
    if (temp == dest)
    {
        string temp = dest;
        while (temp != "\0")
        {
            str_arr[cnt++] = temp;
            temp = mp[temp];
        }
        cout << endl << "DFS的搜索顺序是：" << endl;
        for (int i = cnt - 1; i >= 0; i--)
            cout << str_arr[i] << endl;
        cnt = 0;
        state = 0;
        return true;
    }

    //pos记录str中'0'的下标
    int pos = str.find('0');
    //利用x和y表示pos在3*3棋盘中的二维坐标
    int x = pos / 3, y = pos % 3;
    if (y < 2)//如果y等于2说明str中的0无法向右交换位置
        if (s_push(str, pos, 1, temp))
        {
            /*此处设置string类型变量t的原因是temp是引用的参数，
            在执行if语句中的dfs(t, depth + 1)后temp会发生变化*/
            /*如果dfs(t, depth + 1)返回false，在后续的mp.erase(temp)中
            无法删去原来的temp，所以需要用t来记录*/
            string t = temp;
            //假设t状态是搜索路径上的状态，将其加入到mp中
            mp[t] = str;
//如果if语句返回true表示t状态确实是搜索路径上的状态，返回true即可
            if (dfs(t, depth + 1))
```

```
                return true;
//如果if语句返回false表示t状态不是搜索路径上的状态，需要将其从mp中删除，
            mp.erase(t);
        }
    if (y > 0)//如果y等于0说明str中的0无法向左交换位置
        if (s_push(str, pos, -1, temp))
        {
            string t = temp;
            mp[t] = str;
            if (dfs(t, depth + 1))
                return true;
            mp.erase(t);
        }
    if (x < 2)//如果x等于2说明str中的0无法向下交换位置
        if (s_push(str, pos, 3, temp))
        {
            string t = temp;
            mp[t] = str;
            if (dfs(t, depth + 1))
                return true;
            mp.erase(t);
        }
    if (x > 0)//如果x等于0说明str中的0无法向上交换位置
        if (s_push(str, pos, -3, temp))
        {
            string t = temp;
            mp[t] = str;
            if (dfs(t, depth + 1))
                return true;
            mp.erase(t);
        }
    return false;
}
int main()
{
    //srand用于伪随机数生成算法播种
    srand((unsigned)time(NULL));
        //cin >> sour>>dest;//手动输入 源状态和目标状态
```

```
//随机生成源状态和目标状态
random_shuffle(sour.begin(), sour.end());
random_shuffle(dest.begin(), dest.end());
//sour = "562734810";
//dest = "156427308";

//sour = "504283716";
//dest = "725681043";

//sour = "273645801";
//dest = "123804765";

cout << endl;
cout << "输入是: " << sour << endl;
cout << "输出是: " << dest << endl;

//BFS初始化操作
Begin = clock();
//将源状态sour的下一个状态设置成0
mp[sour] = "\0";
//将源状态sour放到队列q中, 作为BFS的初始条件
q.push(sour);
while (!q.empty())//BFS
{
    string str = q.front();
    q.pop();
    int pos = str.find('0');
    int x = pos / 3, y = pos % 3;
    if (y < 2)//如果y等于2说明str中的0无法向右交换位置
        q_push(str, pos, 1);
    if (y > 0)//如果y等于0说明str中的0无法向左交换位置
        q_push(str, pos, -1);
    if (x < 2)//如果x等于2说明str中的0无法向下交换位置
        q_push(str, pos, 3);
    if (x > 0)//如果x等于0说明str中的0无法向上交换位置
        q_push(str, pos, -3);
    if (state)
    {
        string temp = dest;
```

```
            /*将temp初始化为dest,然后将temp存到数组str_arr中,
            再将temp设置为其前一个状态mp[temp]*/
            while (temp != "\0")
            {
                str_arr[cnt++] = temp;
                temp = mp[temp];
            }
            cout << endl << "BFS的搜索顺序是:" << endl;
            for (int i = cnt - 1; i >= 0; i--)
                cout << str_arr[i] << endl;
            cnt = 0;
            break;
        }
    }
    //执行完while语句之后如果state还是0的话说明查询失败
    if (state == 0)
        cout << "BFS not find" << endl;
    End = clock();
    //使用double(End - Begin) / CLK_TCK表示从Begin到End的运行时间
    cout << "BFS cost " << double(End - Begin) / CLK_TCK *1000;
    cout<< "ms" << endl;

    //DFS初始化操作
    Begin = clock();//重新给Begin赋值
    state = 0;//将state重置为0
    mp.erase(mp.begin(), mp.end());//将mp容器清零
    mp[sour] = "\0";//将源状态sour的下一个状态置为0

    /*调用dfs函数,如果成功则直接退出循环;如果max_depth增加到了25
    还没有成功,则认定为查询失败,退出循环*/
    while (!dfs(sour, 0) && max_depth < 25)
    {
        max_depth++;
        mp.erase(mp.begin(), mp.end());//将mp容器清零
        mp[sour] = "\0";//将源状态sour的下一个状态置为0
    }
    if (max_depth == 25)
        cout << "DFS not find" << endl;
    End = clock();
```

```
//使用double(End - Begin) / CLK_TCK表示从Begin到End的运行时间
cout << "DFS cost " << double(End - Begin) / CLK_TCK * 1000 ;
cout<< "ms" << endl;

//DFS初始化操作
Begin = clock();//重新给Begin赋值
//将源状态sour放到优先队列pq中，作为Heuristic search的初始条件
pq.push(pair<int, int>(0 + diff(sour), 0));
mp.erase(mp.begin(), mp.end());//将mp容器清零
mp[sour] = "\0";//将源状态sour的下一个状态置为0
step[cnt] = 0;//此时cnt为0，0表示源状态的下标，将源状态的步数置
str_arr[cnt++] = sour;//将源状态sour存到str_arr数组中
/*Heuristic search整体与BFS相似，不过q换成了pq，并且pq中记录的
不是状态值，而是状态的估价函数值和状态的下标*/
while (!pq.empty())
{
    /*second表示pair<string,string>的后一个值，定义pq时
    表示该状态在str_arr和step中的下标*/
    int p = pq.top().second;
    //利用p获取str
    string str = str_arr[p];
    pq.pop();
    int pos = str.find('0');
    int x = pos / 3, y = pos % 3;
    if (y < 2)//如果y等于2说明str中的0无法向右交换位置
        pq_push(str, pos, 1, step[p] + 1);
    if (y > 0)//如果y等于0说明str中的0无法向左交换位置
        pq_push(str, pos, -1, step[p] + 1);
    if (x < 2)//如果x等于2说明str中的0无法向下交换位置
        pq_push(str, pos, 3, step[p] + 1);
    if (x > 0)//如果x等于0说明str中的0无法向上交换位置
        pq_push(str, pos, -3, step[p] + 1);
    if (state)
    {
        cnt = 0;
        string temp = dest;
        /*将temp初始化为dest,然后将temp存到数组str_arr中，
        再将temp设置为其前一个状态mp[temp]*/
        while (temp != "\0")
```

```
                {
                    str_arr[cnt++] = temp;
                    temp = mp[temp];
                }
                cout << endl << "HS的搜索顺序是：" << endl;
                for (int i = cnt - 1; i >= 0; i--)
                    cout << str_arr[i] << endl;
                cnt = 0;
                break;
            }
        }
        if (state == 0)
            cout << "HS not find" << endl;
        End = clock();
        //使用double(End - Begin) / CLK_TCK表示从Begin到End的运行时间
        cout << "HS cost " << double(End - Begin) / CLK_TCK * 1000 ;
        cout<< "ms" << endl;
        state = 0;
        return 0;
}
```